THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 56, No. 7, September 1977
Printed in U.S.A.

No. 4 ESS:

Maintenance Software

By M. N. MEYERS, W. A. ROUTT, and K. W. YODER

(Manuscript received July 21, 1976)

To ensure quality service, a toll switching system must be able to meet very stringent dependability and maintainability requirements. To meet these requirements a large package of maintenance software has been developed. This software consists of four functional areas. The first area deals with the detection and recovery from software malfunctions. These malfunctions include failing defensive program checks, scheduling irregularities, and mutilated data. The second area is concerned with the recovery from hardware faults. Hardware fault recovery is stimulated by a failing hardware check and is completed when the system has been reconfigured around the faulty unit. The third area is concerned with the diagnostic programs that aid the craftsperson in the identification and repair of the faulty unit. The fourth area provides for overall coordinated system recovery from multiple or very severe hardware and software malfunctions.

I. INTRODUCTION

To ensure quality service, a toll switching system must be able to meet very stringent dependability and maintainability requirements. Dependability requirements are defined in terms of service continuity and accuracy. Maintainability requirements provide a measure of how quickly hardware or software malfunctions must be corrected. These requirements of dependability and maintainability considerably influence the design of the hardware and software subsystems composing No. 4 ESS.

Continuity of service is provided by both hardware and software redundancy. Hardware redundancy takes the form of providing mechanisms to switch to a spare unit whenever a hardware fault is detected.

The spare unit then performs the function of the faulty unit during the repair process. Redundant software is provided by mechanisms that regenerate program and data structures found to be in error.

Accuracy of service is guaranteed by extensive checking mechanisms in both hardware and software. Typical hardware checks include parity and matching circuits. Software checks include the auditing of data structures and program sanity. The failure of a hardware or software check provides the stimulus for switching to the spare unit or for regenerating a data structure.

Hardware and software maintainability is provided through tools that aid in the rapid repair of system faults. Hardware repair aids include extensive on-line diagnostic tests that isolate a hardware fault to a small number of replaceable circuit packs. Software repair aids include output messages that contain the necessary data to aid in the isolation of the software fault to a particular program or data structure.

The above basic plan is based on prior experience with electronic switching systems.[1,2] However, new concepts and significant extensions of the prior art have been incorporated throughout the design.

1.1 Maintenance software

To meet the stated dependability and maintainability requirements, a large package of maintenance software has been generated. This software has been functionally divided into four areas. The first area is concerned with the detection and recovery from software malfunctions. These malfunctions include failing program checks and illegal data structures. The second area provides for the recovery from hardware faults. Hardware fault recovery is stimulated by a failing hardware check and is completed when the system has been reconfigured around the faulty unit. The third area is concerned with the diagnostic programs that aid the craftsperson in the identification and repair of the faulty module. The fourth area provides for overall coordinated system recovery from multiple hardware and software malfunctions.

II. SOFTWARE ERROR RECOVERY

No. 4 ESS depends upon the data contained in its memories to control the actions of the system. Also, in an operational mode, No. 4 ESS can write into any of its memories and consequently the system is subject to memory mutilation. Therefore, it is necessary to make the system as error-tolerant as possible and also as error-free as the architecture will permit. In order to be error-tolerant, the system must operate in the presence of memory mutilation. In order to be as error-free as possible, there must be restrictions placed upon the software system which can be learned from the analysis of previous systems' error characteristics.

Once the error characteristics are defined, one can strive toward error prevention. Knowing the system can never prevent all errors from occurring, one attempts to achieve a system that is tolerant of as many errors as possible. Then, for the types of errors the system is not tolerant of, error-detection schemes must provide rapid detection. Once errors are detected they should be handled efficiently to minimize real-time usage and to assure the integrity of No. 4 ESS.

2.1 Software-error characteristics

In order to achieve error tolerance and error prevention in No. 4 ESS, software errors must first be defined and characterized. An error can be defined as any data that cause the system to operate abnormally.

2.1.1 Causes of errors

Errors can be introduced into the system in many different ways. In No. 4 ESS, it is the intent of the software-error recovery strategy to eliminate or reduce the occurrence of as many causes of software errors as possible. There are two causes of errors, however—hardware faults and craftsperson errors (other than those at the man-machine interface)—that are not considered within the scope of this section on software-error recovery.

Often programmers have to be aware of and understand many complex and nonstandard program interfaces. The lack of this understanding often results in programs that cause errors when communicating with other programs.

A programmer who is not fully aware of or does not understand the system rules can produce programs that violate one or more of these rules. This can result in data mutilation. For example, a programmer who does not know that the system's shadow registers are destroyed during certain subroutine calls can write a program that leaves pertinent data in the shadow registers over one of these subroutine calls. The information held by these shadow registers will therefore be destroyed upon the subroutine's return.

Logic or coding errors can cause data mutilation. It is possible for these types of errors to go undetected by the program debugging processes, especially if the errors reside in an infrequently entered path of the program. The following two examples draw a distinction between a logic error and a coding error. A logic error would be using 2*(base + index) to derive an address when base + 2*index should have been used. A coding error occurs when the programmer codes an SWK instruction when CWK was intended.

Man-machine interface procedures that are complex and nonstandard often cause errors. For example, if a problem exists that requires per-

sonnel on duty to follow a poorly defined or overly complex procedure at the Master Control Console (MCC), confusion will often result. This will frequently lead to erroneous action at the MCC, thus compounding the problem.

2.1.2 System effects from errors

System effects from errors can appear in several different ways.

The most severe effect of an error is the loss of processing viability, where the processor does not have the ability to perform any software functions—that is, an error which causes a loss of program sequencing such that the system is driven into a system initialization phase. (This kind of error is discussed further in Section V.)

Even though loss of processing viability results in the loss of call-handling capability, it is possible to retain viability (do other work and cycle) and yet have no calls completed through the system.

A facility can be considered as any equipment or software item required for proper system operation. Specific facilities are required to perform a given function. An error that causes denial of a facility will affect the system by restricting the associated function. The error effects that are characteristic of this category do not include a denial of the total facilities that constitute a function.

Loss of a function is closely related to the previous one, denial of facilities. Facilities are required to perform a function. Therefore, an error which causes the total loss of one type of facility implies the loss of the associated function. The loss of a function can also be caused by a scheduling error that does not allow the function to ever be entered.

The capacity of an office can be significantly reduced as the result of system errors. For example, if a link word is "garbaged" part way down a link list of idle call registers, which are required on all calls, then the office has a reduced number of call registers to work with. Therefore, the load-handling capacity of the office has been reduced.

Loss of a single call is the result of multilation of information pertinent to the setup of a single call. For instance, destroying digits in a call register will cause the aborting or mishandling of the associated call. If, however, similar data associated with many calls is consistently multilated, then the system effect will clearly be more severe.

It is also possible for errors to have no effect on the system. One example is mutilation of a word in an unassigned trunk register.

2.2 Impact of software-error prevention and toterance

To ensure the integrity of the memory in the No. 4 ESS, the first step is to prevent the occurrence of as many errors as possible. But errors will still occur. Therefore, to further ensure the integrity of the memory, the system should be as error-tolerant as possible.

2.2.1 Error prevention

After the causes of errors were considered, three general methods of error prevention became apparent. These were standardization, simplification, and improved documentation.

These general methods of error prevention were applied to potential causes of errors in No. 4 ESS. Increased standardization was directed toward program interfaces and man-machine interface procedures. Simplification was applied to program interfaces, man-machine interface procedures, and main call flow under overload. And last, documentation improvement was applied to the areas of system rules and man-machine interface procedures.

2.2.2 Error tolerance

After error prevention techniques were applied to No. 4 ESS, attempts were made to improve the error tolerance of No. 4 ESS, since errors will still occur. Error tolerance implies that the system is able to operate in the presence of memory mutilation.

In an attempt to achieve a high degree of error tolerance, two major mechanisms are used in No. 4 ESS. These are defensive coding and defenses for memory.

Defensive coding mechanisms are used when writing programs in an attempt to remain operational in the presence of errors. When this is not possible, the goal is to operate so that an error will have a minimal effect on the system. In order to operate properly in the presence of errors, defensive coding in programs attempts to detect any error in the data that the program is using. In order to operate with a minimal effect on the system, while in the presence of undetected errors, defensive coding in programs attempts to restrict program accessing of data. It attempts to restrict access to noncritical areas where memory mutilation will have a minor effect on the system.

Specific types of defensive coding that were used towards both of the above-mentioned objectives are:

(*i*) Checking state codes
(*ii*) Range checks
(*iii*) Positive decisions (no decisions by default)
(*iv*) Symbolic addressing
(*v*) Interpreting all possible stimuli
(*vi*) Linking by index (rather than link by absolute address)

In No. 4 ESS there is certain information aside from the actual programs stored in writable memory that is critical to the proper operation of the system. Critical memory can be considered as any memory in which an error, if it occurs, could have a drastic effect on the operation

of the system. An example of critical memory is the basic office parameters (e.g., starting addresses of software items, numbers of active hardware unit types, etc.). If an error occurred in these parameters the operation of the system would be severely affected. Therefore, it is essential to protect this critical memory.

The one main defense used for critical memory is a physically protected area. A special instruction is required to write into the protected area. Therefore, the physically protected area of memory is guarded against wild writes. Also, physical protection becomes an even more powerful defense when the frequency of writing in the protected area is kept very low.

No. 4 ESS, having writable program stores, requires some form of protection for them also. Therefore, no operational programs are allowed (as a standard procedure) to write into program store except for the paging routine and specific recovery routines. In addition, the program stores are all physically protected as described above.

It should be noted that duplication of memory is not considered a defense since it protects only against hardware faults and not against memory mutilation resulting from erroneous software operations.

Another defense for memory is to provide a software protection scheme when operation programs are writing disk. The disk system contains backup copies for critical memory and program stores in addition to storing noncritical disk-only data. Therefore, some protection is required when operational programs are writing into noncritical disk areas. The software protection scheme checks identification tags on write requests against a list of valid writable areas for that given identification tag.

The third form of defense for memory is defensive memory layouts in the unprotected area. Even though the most critical office information is stored in the protected area of memory, defensive memory layouts still need to be employed in the unprotected area of memory. There is still important office information stored in this area which, if mutilated, could have a serious effect on the operation of the system. Since there is a high frequency of writing in the unprotected area, there is a high probability of memory mutilation occurring there. The two major methods used in unprotected memory are to disallow any common scratch areas, and to prevent overlapping of private scratch areas by requiring all private scratch to be allocated by COMPOOL (common pool of data) and defined on COMPOOL.

2.3 Software structure

Once all error-prevention and error-tolerance measures were taken, a software structure was designed to detect, analyze, and correct the

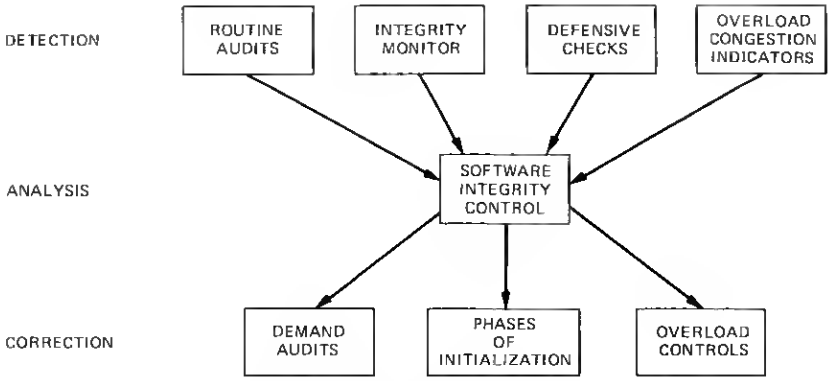

Fig. 1—Software integrity control function.

software errors that will still occur. The software structure that exists for software-error recovery has three major components. These are:

(*i*) The software integrity control program
(*ii*) The audit system
(*iii*) The software-integrity monitor system

2.3.1 Software integrity control

The Software Integrity Control (SICO) program serves as the centralized control for the integrity function. It has all software errors reported to it, makes decisions about the appropriate actions needed, and then activates the appropriate corrective action (see Fig. 1).

The detection of software errors is done via the audit system, the integrity monitor system, and defensive checks implanted throughout the entire No. 4 ESS software. The audit system detects primarily data errors, the integrity monitor detects primarily scheduling and cycling irregularities, and the defensive checks detect primarily mutilated data. All of these errors, when detected, are reported to the SICO program for analysis and corrective action. After analysis, SICO can decide whether to request an audit to correct the error, and if so, which audit. SICO can also make a decision, based on an audit history, to escalate the request to a more severe corrective action such as a phase of software initialization (Section V).

In addition, SICO also receives reports on internal machine congestion from the overload program. This permits SICO to check via other error reports whether this was a congestion falsely indicated by software errors. If so, SICO will request an audit to correct the error, otherwise SICO will allow the overload program to activate the appropriate overload controls.

2.3.2 Audit system

The audit system detects and corrects software data errors. It is basically composed of a control structure and several audit routines, each one tailored to a specific data structure or a group of similar data structures.

The audit control structure schedules the routine audits in addition to running demand audits. The routine audits are run according to the direct search method of error detection. The frequency for running each routine audit is dependent upon the system's sensitivity to errors in that particular data structure or group of data structures. Therefore, the more critical audits are run at a higher frequency than the less critical audits. Also, the audit control structure interleaves the running of routine audits so that the longer duration audits do not lock out the shorter-duration and usually more critical audits. The demand audits, requested either manually or automatically (via the SICO program), are run on a higher priority than routine audits and are not interleaved.

The audit routines detect errors using three basic techniques. These are:

(*i*) Direct comparison (e.g., comparing data with a duplicate copy in core or on disk).

(*ii*) Comparison by association (e.g., verifying that the proper registers are linked together).

(*iii*) Format comparison (e.g., verifying that the data in a particular register appears to be reasonably correct).

The individual audits are written for a specific data structure or a group of similar data structures. The general types of data structures audited are common usage registers, timing structures, queues, and general lists.

2.3.3 Integrity monitor system

The integrity monitor system is primarily concerned with detecting scheduling and cycling irregularities as well as losses of major system functions. It is composed of the general time monitors, the software integrity monitors, and the test call program.

The first of the time monitors is the Program Sanity Timer (PST). The PST is a hardware timer in the central control with a time-out interval of 640 milliseconds (ms). Within this interval, an enable signal must be sent after 320 ms have elapsed. A reset signal must be sent after the enable and before time-out. The PST is administered by the system integrity monitor program on interject. The fact that the system does not time out verifies that the system software has a certain amount of competence or sanity. If the PST times out, a B-level interrupt is generated and a phase of software initialization is run.

The second time monitor is the K-level interrupt. A K-level interrupt occurs whenever the 10-ms clock attempts to set the software interject request flag and the flag is already set. This situation occurs when interject has not been served for 10 to 20 ms. When a K-level interrupt occurs, a failure count is incremented and compared with a threshold. If it exceeds the threshold, a phase of initialization is run via the SICO program. If the threshold is not exceeded, the interrupt returns to normal processing.

The Software Integrity Monitor on Base level (SIMB) performs detailed checks to verify the validity of the base cycle. This includes checking base-level program entry counts, comparing the base-level cycle length just completed with the previous cycle, and comparing the last base-level cycle length with a minimum allowed value. SIMB also routinely performs functional checks to verify that major system functions such as the disk system and the input/output system are available.

The Software Integrity Monitor (SIM) on interject checks the validity of the scheduling on interject and ensures entry to other integrity routines lower in the scheduling structure. SIM also, as previously mentioned, administers the PST. Both SIMB and SIM use failure counters and appropriate threshold values when deciding whether to report a failure to the SICO program.

The test call program provides a gross check upon the system's operation by preventing special calls to the system and observing the progress of these calls. If a call should fail to progress as anticipated, then checks are made which attempt to isolate the cause of the trouble. The test call program consists of four sections: a generator, a call monitor, a progress monitor, and failure processing. The test call program presents calls of all three pulsing types: MF, DP, and CCIS.

III. HARDWARE ERROR RECOVERY

The nature of No. 4 ESS peripheral hardware and the software strutures used to control it are the two major points related to hardware-error recovery. The hardware is highly autonomous. Various means for providing redundancy and error detection are used. The separate hardware units are tied into an overall interrupt structure.

The software controls the interconnection of communication and control paths between the peripheral units and the central processor, and between peripheral units themselves.

3.1 Hardware architecture

The periphery of No. 4 ESS can be broken down into three areas: switching, network, and transmission/switching interface, Fig. 2. The last two are often inseparable when error detection and recovery are

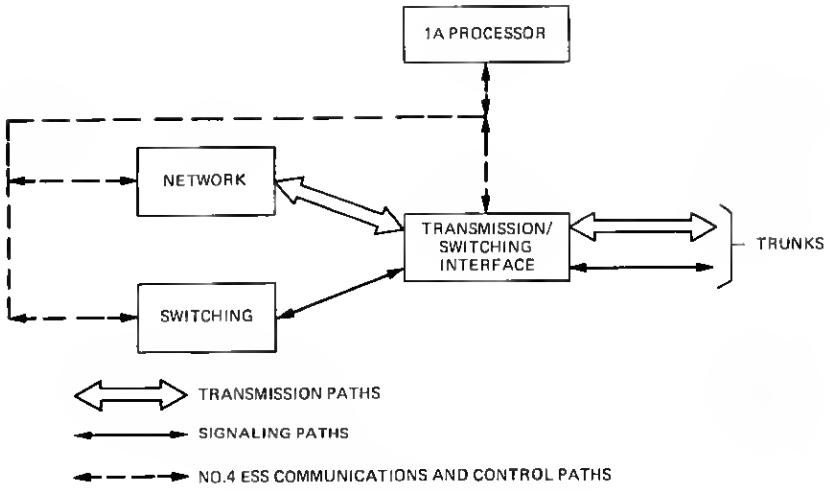

Fig. 2—No. 4 ESS architecture.

considered. However, each of the three areas has a high degree of autonomy and its redundancy and error-detection scheme is unique.

3.1.1 Autonomous nature of hardware

The signaling units (signal processors and CCIS terminal) are autonomous processors. The SP is a wired-logic machine that scans for supervisory changes, and collects/transmits dial pulse and MF signaling information. The CCIS terminal is a programmable processor that performs analogous tasks in its environment.

Each has an independent clock and except for inquiries from the central processor, they are independent of the central processor.

Network and transmission/switching interface frames are linked together by a common network clock and by common transmission paths. These units work together to autonomously set up paths through the system and to convert between various transmission formats (i.e., analog to digital, digital to digital, digital to analog).

The central processor's only contact with these units in the operational environment is to provide path setup information. The time-shared paths are set up and removed independent of further intervention by the central processor.

3.1.2 Redundancy

The redundancy imposed on a peripheral unit is related to the number of trunks affected by a failure, the probability of a fault in the unit, and the practicality of a particular redundancy plan.

Table I — System outage for a total unit failure

Failed unit	Number of trunks or percent of capacity lost	Unit redundancy scheme
Peripheral unit bus	All trunks, 100 percent	Full duplication plus dispersion
Network clock	All trunks, 100 percent	Two fully duplicated halves
Time-multiplex switch	25 percent to 50 percent	Full duplication
Time-slot interchange	1680 trunks	Full duplication
Signal processor	4096 trunks	Full duplication
CCIS terminal access circuit	16,000 trunks	Full duplication
CCIS terminal	10,000 trunks	Load sharing between two terminals
Voiceband interface controller	840 trunks	Full duplication
Voiceband interface unit	120 trunks	Protection-switched space for 7 units
Digroup terminal controller	960 trunks	Duplication (some maintenance features are not fully duplicated)
Digroup terminal unit	120 trunks	Protection-switched space for 8 units
System clock	0 trunk	Full duplication (a software timer is available to back up the system clock)

The need for more or less hardware to perform a chosen function, the economies of scale, and packaging constraints can have more influence on redundancy than the number of trunks affected by an error in the unit. However, in the case of No. 4 ESS, the number of trunks affected by a failure played the major role in determining a redundancy plan.

The loss of the network clock means the outage of the entire No. 4 ESS as a switching machine. The network clock forms the foundation for the entire network and transmission/switching interface. It is a dual-duplex arrangement. There are two pairs of clock chains. Either chain in a pair can fully replace its mate, but the members of one pair cannot take the place of the members of the other pair. A pair provides clocking to one half of all duplicated units in the network and the transmission/switching interface equipment. The other pair provides clock to the other halves.

A total hardware failure of a unit other than the peripheral bus or network clock affects less than 100 percent of the No. 4 ESS capabilities either in capacity or trunk access. The number of trunks that can be denied access or the reduction in capacity that can occur when a unit fails is summarized in Table I.

3.1.3 Error detection

Error detectors provide a means of identifying when errors occur and, if possible, pinpointing the cause of the error. In the No. 4 ESS pe-

riphery many types of error detection are used: parity checks, code checks, $1/n$ enable checks, matching, etc.

These error checks can be classified as unique and nonunique. Unique error detectors indicate the presence of an error and locate the error to a reconfigurable block of the system. A reconfigurable block is half of a duplicated unit, one of n units with a protection-switched backup, etc., that can be placed in or out of service. Sometimes many of these are in series. An example might be the central processor (CP), peripheral unit bus (PUB), and a peripheral unit.

Figure 3 has six configurable blocks or three pairs of interchangeable blocks. A unique error detector would be one that uniquely identified one of these blocks as faulty. If Unit 0 had internal memory with a built-in parity check P, a parity failure would be unique.

Nonunique error detectors identify error conditions but give little information as to which configurable block is at fault. An example is a matcher "m" between an output register in each half of the unit. If a mismatch occurs, no clue exists as to which half of the unit is at fault.

Furthermore, if data input to the unit has any effect on outputs and was not error-checked on the buses, the CPs could be at fault and the error would not be detected until the mismatch occurs.

No. 4 ESS employs both types of error detectors. Most communication paths and links in the periphery employ unique error detectors or those approaching uniqueness.

Unique error detectors proved to be too expensive in most areas where logical and arithmetic operations are performed. In these cases matching was employed between duplicate halves for error detection and resulted in nonunique error detection. When an error is detected in the periphery, the main program in the central processor is notified via an interrupt structure.

3.1.4 Interrupt structure

Errors detected in the periphery are reported to the central processor via hierarchical interrupt levels. There are three levels based on the urgency of correcting the effects of the error.

The F-level interrupt is the highest level for the peripheral system and causes the central processor to give immediate attention to the error condition.[3] It breaks into the basic task being executed and the task may be aborted. An F-level interrupt has two subclassifications, a Peripheral Unit Failure (PUF) and an Autonomous Peripheral Unit Failure (APUF). PUF can be triggered only by error-check hardware when the central processor is actively addressing a peripheral unit via the peripheral bus. APUF can be triggered by error-check hardware when the central processor is not addressing the periphery or when the error is independent of central processor access.

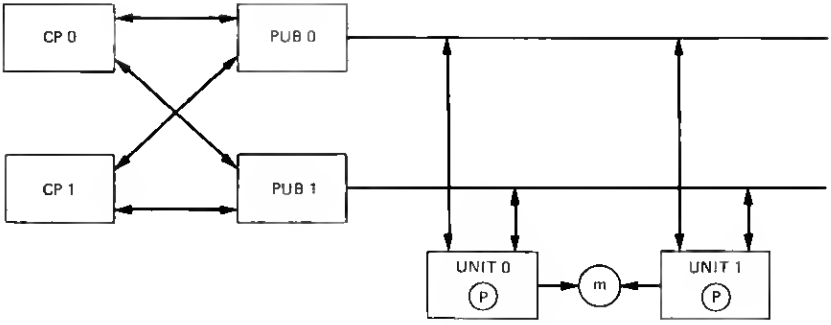

Fig. 3—Example of reconfigurable blocks.

An Autonomous Peripheral Unit Trouble (APUT) is the second level the central processor will recognize. It will be recognized within 3 ms of error detection or as soon as the main program completes its present basic task and arrives at a safe point. Interwrite problems might occur if it were immediately recognized by breaking into the execution of the current task. Where time allows, the autonomous peripheral unit trouble interrupt is used instead of the F level so the current task can be successfully completed.

The Autonomous Peripheral Unit Base-level APUB interrupt is the third and final level. Treatment of an error is deferred even longer, up to 100 ms. The treatment becomes a base-level task and the actual interrupt does not affect the performance of the main program.

Each hardware-error detector causes one of these interrupt levels to be entered, which in turn leads to a software structure that will locate the source of error and isolate it from the active system.

3.2 Software architecture

The software structure which deals with errors consists of functions that are unique to a particular peripheral unit type (concentrated in per-unit type software packages) and functions which control a large subset of the peripheral unit types.

3.2.1 Concentrated Unit Structures

Although most of the peripheral units are connected to the central processor over a common peripheral bus, internally they differ markedly. Even the bus interfaces, although functionally equal for operational access from the central processor, are different when examined in detail. Thus the routines needed to deal with a particular unit type are concentrated in one software package. This allows for better manintainability of these functions because the individual programmer need only

be aware of the detailed workings of one or two units. The types of routines concentrated on a unit basis fall into three broad categories, unit fault recovery, unit bootstrap, and unit configuration.

Unit fault recovery isolates a fault to a configurable portion of a unit type once the source has been traced to a unit. It assumes the central processor and peripheral bus have been exonerated by preceding recovery action. The unit fault recovery filters out the most likely source of error according to a priority structure based on the architecture of the unit. In the case of a nonunique error it will run tests to determine the configurable portion of the unit that is at fault. Once the portion at fault is identified, fault recovery selects a course of action. It then confers with a centralized error analysis to have the decision accepted or changed with regard to previous errors from the same unit or units interacting with it. Upon return from error analysis, unit fault recovery carries out the action agreed upon by setting up intraunit functions itself and by going to a centralized peripheral configuration program to set up interunit functions. Once the action is complete, a report of the error and its resolution is made and all collected data is archived via the 1A common error analysis program.[3] Control is then passed to a system restart program.

Unit bootstrap routines perform initialization of a unit. They assume the unit can be in any state and will bring it to a state suitable to begin call processing. An access test is performed on the unit to ensure that it has basic sanity and that the risk of introducing a formerly out-of-service unit into the overall system is minimal. All of these types of routines—bootstrap and access test—are tied together by a centralized peripheral hardware recovery.

Unit configuration routines provide both inter- and intra-unit routing of communication and control paths. These routines are linked into a common peripheral configuration package by a centralized configuration control program.

3.2.2 Centralized Control Structures

The centralized control structures in peripheral maintenance can be divided into five categories:

Hardware recovery
Peripheral configuration
Peripheral error filtering
Error analysis
System restart

Hardware recovery is called by system recovery (Section V) to select and certify a working combination of peripheral equipment. Each of several levels of hardware recovery is progressively more severe. The first

level tries not to disturb any hardware in service at the time it is entered. Successive levels simplex the equipment and interchange redundant portions of equipment previously left out of service.

At the most severe level, a minimal set of peripheral equipment is reinitialized and configured. The object is to eliminate more and more possible sources of system upheaval that may have lead to system recovery action. To perform its task, hardware recovery calls on unit bootstrap and access test routines as well as unit configuration routines via the peripheral configuration.

The peripheral configuration program acts as a clearing house for all interunit configuration changes of communication and control paths. It calls upon unit routines to perform specific tasks, and it stitches these tasks together to assure access to and from the various areas of the system is not lost to call processing. An example is the removal of a peripheral bus which interconnects the periphery with the processor. Before the bus can be removed, all other units interfacing with the bus must be examined and possibly reconfigured to ensure all in-service units have functional interunit address and control paths with the remainder of the system.

System peripheral-error filtering resides in a program that is entered for each type of error detected from the periphery or at the processor-to-periphery interface. It determines the peripheral unit implicated by the error indicator and then isolates the cause of error to the processor, the peripheral bus, the implicated unit's bus interface, or the implicated unit. It thus must deal with all units in at least a superficial manner. If it determines the error is within the unit, it will transfer control to a unit fault recovery routine for further error resolution.

Error analysis adds the element of past history to fault recovery. It acts to resolve interframe errors that are not associated with the peripheral bus. It maintains a record of all errors and their resolution for a period of time. A decision made by a fault recovery routine is passed to error analysis for examination before it is acted upon. Error analysis can concur or alter a decision, according to past history as examined via a sequence table. Sequence tahles are a collection of decision schemes which make different decisions on successive occurrences of an error. A sequence table is selected by fault recovery for each type of error. If there is not past history active in the error analysis data base associated with the error and unit under investigation, the first decision scheme of the sequence tahle is used to carry out the analysis. If there is past history present, the next decision scheme of the sequence table recorded in the last record of past history is used.

These decision schemes can draw on several factors such as:

The environment of the configurable portion of the system in error (i.e., duplex, simplex etc.).

The number of times the error has occurred over an interval of time.

The type of error (unique, nonunique).

The characteristic of the error (transient, hard failure, illegal system action).

System restart provides a point where all error treatment is terminated and the system is gracefully returned to call processing. Cleanup of the system software resources disturbed by the error is done at this point. Records of the error and error treatment are transferred to an archival data base where they can be retrieved from and analyzed off-line. An attempt is made to restart call processing at a point where as little system perturbation as possible will occur. This is often a difficult task. Errors can occur in a variety of places and the disturbing effects of an error are difficult to predict.

3.3 Fault recovery strategy

Fault Recovery (FR) must change a system partially or wholly incapacitated by an error, believed caused by malfunctioning hardware, into a completely viable system that has shaken off the effects of the error.

FR avoids call disturbances perceptible to the customer. This requires that FR operate within timing constraints dictated by call processing. Only in exceptional cases does FR take excessive time and cause perturbations in call processing.

Errors must be detected as close to their source as possible. The further they propagate, the harder it is to find the source, the harder it is to discriminate the types of error, and the harder it is to clean up the deleterious effects of the error. Within economical constraints, error detectors in No. 4 ESS were placed in the system to provide detection as soon as possible.

Each error indicator and each error source leading to an error indicator has been given a position in an error priority structure. Priority is given to errors occurring during processor access of the periphery, over autonomous errors (see Section 3.1.4). Priority is given to unique error indicators, which allow for the fastest and most concise resolution, over nonunique error indicators.

Once an error indicator is chosen, the error must be classified. Classifications of errors include: software, transient, hard fault, and error analysis resolvahle. Software errors occur when nonexistent hardware is accessed. Transient errors are those which are not repeatable via retries or other techniques. They may be caused by marginal hardware failures, systems noise, etc. Faults cause errors which will be repeated until the fault is removed. Error-analysis resolvable errors are those that cannot

be more precisely classified because of the nature of the error or inadequacies of FR.

The best of several techniques is chosen to identify the portion of the system that caused the error and to classify an error. The simplest technique is to assume the portion of the system containing a unique error indicator is at fault. Other techniques include retries of the sequences of events leading to the error, testing of the hardware involved using test data derived from the data present in the vicinity of the error, and testing of the hardware involved using fixed predetermined test data. Unique error resolution is much preferred over fixed-data testing. It is less time-consuming and more reliable as an identifier of the type and source of the error. Testing with predetermined test data can be likened to a minidiagnostic that runs unsegmented in real time. FR requires resolution to a configurable portion of the system and not to a replaceable module.

Once the type of error is ascertained and the reconfigurable portion of the system with the fault is identified, FR selects a course of action and a sequence table. This information is passed to error analysis. Error analysis will agree with the action or provide an alternate course of action based on the present error and consideration of its relation to past errors. FR will carry out the action finally agreed upon. All data collected during the treatment of the error is archived and the system is returned to normal processing.

3.4 Example

The following is an example of what might happen if an interrupt occurred on an access of a duplicated signal processor's trunk status memory.[4] The central processor would be interrupted by an F level and control would be given first to the routine for peripheral error filtering. This routine would identify the source as failure of the central processor to obtain an All-Seems-Well (ASW) signal from the peripheral unit. Furthermore, it would identify the unit as a particular SP from peripheral address information saved by the processor at the time of interrupt. The SP's routing and error-source registers would be read and saved if the error condition permitted. Then a series of access tests on the SP in question would be executed to verify that the central processor and the peripheral bus were not at fault. If they were certified as good and examination of the SP's internal error-source indicators did not implicate the bus, further processing of the error would be turned over to the SP's unit fault recovery routine (SPFR).

SPFR would examine the SP's error source indicators and for our example find a number of mismatch indicators (internal sequencer mismatch, memory address and data mismatch). The sequencer mismatch is the highest priority of these mismatches and is treated by retrying the

failing order independently on each half of the duplicated SP. Let us assume the retries are inconclusive and SPFR has taken all the time it can. SPFR then classifies the error as error-analysis resolvable, picks a half to be removed from service and diagnosed, chooses a sequence table, and consults with error analysis.

For our example, error analysis already has a similar error for this particular SP on record and notes SPFR chose the same half for removal the last time. Error analysis then changes the decision to remove the other half and SPFR carries out the decision. All data gathered about the error as well as the action taken is archived and the system is returned to normal processing. The SP diagnostic subsequently finds a faulty pack in the logic that causes the two halves to execute peripheral orders in synchronization and the repair is made. Some time later, after the SP has run in full duplex for a period of time, the error analysis past history for this event and the events leading to it are automatically removed by the system from the current error analysis files.

3.5 Expected results

The expected results of hardware recovery are:

(*i*) One interrupt to recover from hard unique errors or software-type errors.

(*ii*) One to two interrupts for nonunique hard errors contained within adjacent units.

(*iii*) Two or more interrupts for error-analysis-resolvable errors, transient errors, and errors with effects propagated over several units along the interunit communication and control paths.

For errors that deviate from these expected results in No. 4 ESS, new or modified sequence tables will be added. Sequence table structures were designed with change in mind. To avoid changing the FR control structures or routines that are inseparable from the hardware they interact with, the decision criteria for treating a particular error source indicator and/or type of fault over time are changed by modifying the error-analysis sequence tables. These tables are a series of macro expansions which can be easily changed with a high degree of confidence that the change is correct and will not cause unexpected side effects on FR recovery from the target errors.

In general, our expectations have been met. Our experience has led to some changes in the original sequence tables to treat high-frequency transients of short durations and to treat errors whose effects propagate further in the periphery than had first been expected. The sequence table structure has been useful in implementing these changes in the decision criteria.

IV. DIAGNOSTICS

4.1 Overview

4.1.1 Objectives

The basic object of a diagnostic program is to detect and locate hardware faults. The diagnostic program accomplishes this objective by:

(*i*) Applying inputs to the unit under test.
(*ii*) Comparing the outputs with the expected outputs in order to detect the fault.
(*iii*) Using the pattern of failing tests to locate the fault.

Typically, a diagnostic program is designed to detect greater than 90 percent of the faults, and for these faults resolve the problem to an average of less than five circuit packs.

Since the repair process is a deferred task involving manual action, the execution time of the diagnostic is not a prime consideration. However, the diagnostic program contains many thousands of tests, and it is desirable to minimize the memory required to store these tests. Thus in No. 4 ESS the diagnostic program is designed to minimize program size at the expense of some execution time.

Unlike most other programs within the system, the diagnostic program listings are used by the craftsperson to manually analyze failure data. The diagnostic programs are therefore designed to be easily read and understood. For ease of use by craft, the diagnostic programs are grouped according to unit type. Within each diagnostic, the tests are subdivided into groups with each group testing well-defined blocks of circuitry.

Since diagnostic programs are often affected by hardware changes, the diagnostic is designed to be easily modified via future generic updates.

4.1.2 Diagnostic execution

A diagnostic program execution can be stimulated from any of several sources. These sources include:

(*i*) Fault recovery following a maintenance interrupt
(*ii*) System recovery following system reinitialization
(*iii*) The craftsperson during the repair process
(*iv*) Routine exercise to perform periodic testing of the frame

Once initiated, the diagnostic program executes concurrently on an interleaved time basis with normal call processing in a noninterfering manner. During diagnostic execution the test results are printed on a teletypewriter. At the conclusion of the diagnostic, a summary message

is printed. This message indicates one of the following: all tests passed (ATP), some tests failed (STF), conditional all tests passed (CATP), or no tests run (NTR). The CATP response is printed whenever it is necessary for the diagnostic to skip tests because of the unavailability of a system resource. Examples of system resources needed by diagnostics include buses, mate units, and pulse points.

If there were any test failures, a list of suspected faulty circuit packs may also be printed at the teletypewriter. Unlike previous ESS, the translation between failure pattern and suspect circuit packs is performed on-line. In all cases, the suspect packs are ordered with the most probable packs printed first. The repair procedure requires sequentially replacing each circuit pack on the list until a diagnostic ATP or CATP condition is reached. The diagnostic program is manually initiated between each circuit pack replacement to check if the fault has been corrected.

4.2 Implementation

4.2.1 Test design language

In order to facilitate the writing of the diagnostic program, a special-purpose language, denoted DIAL, was developed. The DIAL language is oriented to the special requirements of diagnostics in the ESS environment. Statements in DIAL can be divided into two classes: testing statements and general purpose statements.

An example of a testing statement is:

```
STM1 TMSOP OPER (READ), OPAD (4TGOP),
                        MASK (4TGM), EXPR (4TGE)
```

In this case, "STM1" is the statement label, "TMSOP" identifies the type of unit being tested (TMS), "OPER (READ)" identifies this as a read from a unit, "OPAD (4TGOP)" is the input to the unit that will elicit the reply, "MASK (4TGM)" masks the reply from the unit to certain specified bits, and "EXPR (4TGE)" is the expected reply from the unit. For a write to a unit without a corresponding read, the mask and expected result field are defaulted. Similar statements exist for each peripheral unit. In addition many statements exist in common for all peripheral units. Commonality of testing statements is enhanced since most peripheral units have the PU bus as their input/output medium.

The general-purpose statements are similar to most other high-level languages. Statements exist to move data in memory, perform arithmetic and logical functions, call subroutines, etc. The language is procedure-oriented in that the total program is subdivided into a set of "phases" and subroutines called by these phases. Each phase has only one entry

and one normal exit point. Each phase tests a functional block of circuitry and the phases are executed in order.

The DIAL general-purpose statements have several unique aspects. The DOLOOP statement allows for the shifting or rotating of specified data patterns each time through the loop. This feature facilitates the generation of test patterns for regular logic. Another unique feature is that program branches are allowed only in the forward direction. This feature facilitates program reading and debugging; however, the main impetus for this restriction is that unique test numbers must be assigned at compile time to each test. If tests are skipped during diagnostic execution, the test number can be advanced correspondingly.

Assembly language coding of diagnostic tests is not allowed and DIAL is generally independent of the 1A Processor. The testing statement parameters are specified in terms of the unit inputs and outputs, not in terms of the 1A Processor. This independence of the host computer facilitates the writing of compilers for other machines. Compilers have been written for No. 1 ESS and a host of various minicomputer systems. These minicomputer systems execute the diagnostic programs in other environments such as in factory frame testing.

A compiler also has been developed to translate the diagnostic program into LAMP logic simulator inputs.[5] This tool made it possible to execute the diagnostic on a software model of the unit under test before the unit was physically available. Many hardware and software design errors were thus detected early in the development.

4.2.2 Test generation and evaluation

The number of tests that must be generated required the development of several aids. One of these aids is DIAL, which allows the tests that follow a repetitive pattern to be easily coded using the DOLOOP and subroutine features of the language. Another aid was the development of a set of programs which would map existing circuit pack tests into the frame diagnostic tests. Automatic test generation programs were also used to generate some of the tests.

The tests are evaluated by both physical fault insertion and the LAMP simulator.[5] Lamp provides a means to verify whether the test will pass on a fault-free machine. This feature is used to debug tests before the actual frame is available. Another feature of LAMP is the ability to measure and identify the number of faults that would be undetected by the diagnostic. This feature provided a measure of diagnostic effectiveness and indicated areas for diagnostic enhancement.

4.2.3 Diagnostic structure

For No. 4 ESS, the DIAL compiler was written to generate a compact representation of the program in a "data table" or interpretative format.

An on-line control program interprets the data table at execution time to effect the execution of the diagnostic.

An interpretative program has another advantage in that other controlling operations can be implemented easily. One example is the automatic segmenting of the diagnostic program. To allow for concurrent execution with call processing the diagnostic execution is broken into time segments of approximately 3 ms. With an interpretative control, this segmenting is done at execution time with such variables as unit response time being automatically accounted for. For those rare cases where the segment boundaries must be explicitly specified, facilities exist in the language to define the segment boundaries at compile time.

Interpretative control also provides for manual interactive control of the diagnostic execution. Features in the interactive control subsystem allow the craftsperson to pause at selected points within the diagnostic execution, loop the diagnostic execution over specified addresses, etc. The input commands, received from the craftsperson by the control program, cause the diagnostic execution to conform to these commands. Automatic segmenting provides advantages to interactive diagnostic use, since the craftsperson can pause or loop the diagnostic virtually anywhere without taking the segment boundaries into account.

The diagnostic control program implementation is similar to that for 1A Processor Units. This commonality provides savings in design effort and forces uniformity of man-machine interfaces. Many features common to 1A Processor and peripheral diagnostics have proved to be very valuable. One common feature of 1A Processor and peripheral diagnostics is the ability to execute several diagnostics concurrently. This feature is especially valuable for peripheral diagnostics because of the large number of peripheral units. Intefering peripheral diagnostics are automatically prevented from executing concurrently.

4.2.4 Diagnostic output

Diagnostic output includes the raw data output of the failing tests and a list of suspect circuit packs. The raw data output includes the following information for each failing test: the unique test number, the test failure pattern, the actual response from the unit on the PU reply bus, the input to the unit on the PU address and write buses, and the location in the diagnostic of the failing test. The purpose of this raw data output is to present to the craftsperson an easily understood description of the test(s) that failed. This information would be used in the manual analysis of data whenever the suspect pack printout failed to locate the problem. Figure 4 shows a sample raw data teletypewriter output.

The listing of suspect circuit packs gives those packs that are most likely to contain the fault. Included in the output is the physical location of the suspect and the circuit pack code. Other special information

```
DGN:TMSP 0, CONTR 1 PH 5 STF (MISMATCHES=3)

TEST          MISMATCH  SUPPLEMENTARY DATA
  17          01000000  01000000  12024540  00774000  00000266

test number  mismatch  response     input to unit      location of
                       from unit                       test failure
```

Fig. 4—Sample raw data output.

concerning the pack, such as warning flags, may also be printed. Figure 5 shows a sample suspect circuit pack list.

4.2.5 Trouble location methods

Several distinct methods have been developed to map the diagnostic failure pattern into the list of suspect circuit packs. One method is an on-line pattern analysis of the failure data. This method is especially applicable to cases where the logic is regular. For example, the address of the failing memory word plus the failing bit will normally uniquely identify the faulty circuit pack.

Another method of fault location is to match the failure pattern with a predetermined set of failure patterns. These predetermined failure patterns are gathered by a combination of physical fault insertion and fault simulation. The algorithm uses seven key parameters derived from the failure pattern in an attempt to attain as close a match as possible. This method is the same as used by most processor units.[3]

However, peripheral units generally use a method based on the circuit

```
M 36 ANALY:TLPFILE  TSIF 0, CONT 0 SUSPECTED FAULTY EQUIPMENT
     TLPFILE 57   ENTRY TIME 01/11/76 23:33:13
     EQPT LOC   CODE   NOTE WT  FS SYM   SD   HELPER ID
     046-19     FA0557       9  11   8  4A024
     042-25     FA0543       9  13   1  4A024
     046-17     FA0540       5   9   2  4A024
     042-21     FA0632       5  10   8  4A024
     042-13     FA0540       5   3   3  4A024
     042-19     FA0551       5  14   2  4A024
     046-18     FA0540       5   9   1  4A024
```

Fig. 5—Sample suspect circuit pack list.

topology of the unit and the diagnostic failure pattern. For peripheral diagnostics, the failure pattern contains the individual bits that failed plus the "ADDRESS" of the point within the unit read for this test. This point (address and bit position) is known as a monitor point. These monitor points are typically flip-flops, internal registers, dc leads, parity bits, etc. The diagnostic failure pattern thus maps into a list of failing monitor points.

The circuit topology of the unit is contained in what is known as a "connectivity data base." This data base is a listing for each monitor point of the circuit packs associated with these monitor points.

The generation of the connectivity data base involves the following operations. First, all monitor points within the unit are identified. Second, a list of associated circuit packs for each monitor point is generated. This list of associated circuit packs is made up of the following two components:

(*i*) All circuit packs containing logic paths to this monitor point from external inputs or from other monitor points.

(*ii*) All circuit packs containing logic paths which transmit the state of this monitor point to external outputs.

Off-line, a list of circuit packs associated with each monitor point within the unit has been generated and stored on tape. This tape is accessed by the ESS resident diagnostic-results processing programs.

First, the on-line fault location procedure summarizes the monitor point occurrences in the failure pattern. Second, the union of the associated circuit pack lists for each failing monitor point is generated. Third, the circuit packs are ordered according to various criteria. Examples of possible criteria include number of occurrences, reliability data, number of gates in the logic path on this circuit pack, etc.

A trouble-locating method based on the circuit topology has several advantages. First, the method is independent of the diagnostic program. One can add tests to the diagnostic without affecting the trouble-locating method. With methods that rely on previously generated fault signatures, test enhancement is difficult since it must not affect the existing fault-signature data base. Second, the connectivity data base can be automatically generated from existing files containing the circuit description. This attribute is important if the trouble location procedure must respond to hardware changes. Other methods restrict such changes or force the regeneration of the data base.

The decision on which trouble-locating method to use is based on several considerations. In general, the decision revolves around the following points. Pattern analysis can be used for regular logic. Methods based on predetermined fault signatures are used for units that must have high trouble-location accuracy and resolution. Methods based on

circuit topology are used for the remaining units. It is also possible for a diagnostic to use a combination of methods based on different considerations within the diagnostic—for example, pattern analysis for the regular logic and connectivity for the irregular logic.

4.2.6 Routine testing

Since up to 25 percent of the hardware circuitry is involved only in maintenance operations, it is important to routinely exercise this circuitry. The usual hardware checks will detect only faults in the operational circuitry. The problem is determining the frequency of routine testing. Infrequent testing increases the possibility of multiple faults. Frequent routine testing decreases reliability by increasing the simplex operation time. For peripheral units, formulas were developed to calculate the optimal frequency of routine testing.

4.3 Results

Significant results were achieved in several areas of diagnostic program design. First, the high-level diagnostic language increased diagnostic programming productivity, standardized the diagnostic design effort, and enabled the diagnostic programs to be compiled into a form for use in several diverse applications. Second, the common structure encompassing both 1A Processor and peripheral diagnostics decreased the program integration effort, provided for a uniform man-machine interface, and led to commonality of design. Third, the various support programs such as LAMP decreased design effort and provided a way to evaluate tests independent of physically inserting faults into the machine. Finally, the use of the connectivity trouble-location methods enabled the generation of quality trouble-location algorithms with significantly less effort than has been applied with other methods.

V. SYSTEM RECOVERY

Memory mutilation in either program store or call store can cause abnormal system operation. Such situations can be detected by monitoring various system characteristics and auditing certain data structures, as discussed in Section 2.3. Once it is decided that severe mutilation has occurred and remedial actions such as demand audits will not correct it, system recovery actions are taken to reconstruct a sane program and data base and to obtain a viable hardware configuration. System recovery basically consists of hardware reconfiguration and software initialization and it can be invoked either automatically under program control or manually.

5.1 Automatic system recovery

Automatic system recovery takes place when the program detects memory mutilation and determines that a severe recovery action is needed. Automatic system recovery is needed when remedial actions such as demand audits are not able to correct the problem causing abnormal system operation. These system recovery actions are termed phases of initialization and these phases increase in the severity of their corrective actions.

5.1.1 Justification

5.1.1.1 Need for system recovery. System recovery in the form of a phase of initialization is needed whenever one of several severe problems has occurred, affecting normal operation. Some of these are:

(*i*) Mutilation of writable program store
(*ii*) Loss of a vital system function
(*iii*) Loss of a major facility
(*iv*) Escalation of remedial actions
(*v*) System start-up

System start-up is not a problem as such, but does require a phase of initialization and occurs whenever the system has been "down" for any length of time or whenever a complete new issue of the program is being loaded.

5.1.1.2 Phase triggers. There are specific triggers built into the No. 4 ESS that will cause a phase of initialization to occur. These phase triggers were chosen in an attempt to clear problems as quickly as possible which were determined to be severe enough in nature that the taking of further remedial actions (or any action in some cases) would only delay their correction. The basic phase triggers are:

(*i*) Problems in the software integrity control program (SICO)
(*ii*) Excessive lower phases (phases 1 and 2)
(*iii*) Program sanity timer time-out
(*iv*) Excessive maintenance interrupts
(*v*) Excessive audit requests
(*vi*) Excessive K-level interrupts
(*vii*) Nonservice of interject programs
(*viii*) Nonservice or mutilation of the software clock
(*ix*) Invalid entry to the interject monitor
(*x*) Duplex failure of a unit

These triggers each request a specific phase of initialization and these requests can be escalated based upon the recent occurrence of other phases of initialization.

5.1.2 Recovery sequence

The sequence of system recovery consists of hardware reconfiguration and recovery as well as software recovery or initialization.

5.1.2.1 Design philosophy. The phases of initialization were designed such that each phase increases in severity. The lowest-level phase was designed to be the least severe and fastest running, and was intended to clear a majority of the problems. Most problems can be cleared by a short, direct phase and do not require a complete system initialization. All phases are designed with the philosophy of initializing memory as opposed to auditing and correcting memory. Initialization is generally faster and more effective at clearing severe problems or memory mutilation than detecting and correcting errors.

The phases are numbered 1 through 4, phase 1 being the least severe. It is short in duration and it assumes all hardware is good and all permanent memory is good. It initializes specific areas of transient memory. The phase 1 also saves all calls.

Phase 2 is next in the escalation order and it assumes all processor hardware is good and all permanent memory is good. It basically reconfigures the peripheral hardware and initializes all of transient memory. It saves stable calls.

Phase 3 assumes nothing is good. It first establishes a processor hardware core and then a complete processor hardware configuration. Next permanent memory is verified and/or reinitialized. The peripheral hardware is configured and then transient memory is initialized. Phase 3 saves stable calls. Phases 1 through 3 can be activated automatically or manually.

Phase 4 is the highest-level phase and it can only be activated manually. It is the same as the phase 3 except that it tears down all calls and as a result totally reinitializes the entire system.

5.1.2.2 Software control structure. The software integrity control (SICO) program controls the execution of system recovery. SICO runs the appropriate hardware and software recovery routines based upon the phase being run and then passes control to the software initialization program (SINT), which performs the initialization of memory and the saving of calls. SICO can also escalate any automatically generated phase request according to upon the trigger and the recent phase history.

Once SINT has completed the software initialization, SICO will once again be given control to prepare the system for restarting after the phase. The duration of the phases are basically dependent upon the phase which is run and the size of the office. Therefore, phases can last from 1 second up to 1 minute or slightly longer. As a result of this outage, certain actions must be taken by SICO before restarting the system, such as clearing out the buffers in the signal processors and instituting overload controls to control the anticipated traffic buildup. SICO also runs

certain specialized restart routines and formats the printout of the phase results.

5.1.2.3 System hardware recovery. As discussed in the previous sections, the processor hardware recovery is accomplished first and is followed by: permanent memory verification and initialization, peripheral hardware recovery, and transient memory initialization (including saving of calls). The processor recovery establishes a hardware core and then basic sanity tests are run on this hardware core. Once these sanity tests are passed, the entire processor hardware complex is established. Permanent memory is then verified by a hash summing procedure and any failing blocks of memory are reinitialized using the backup copy on the disk file system.[3]

Peripheral hardware recovery is then run in four levels. Successive levels increase in severity. The level run depends upon the phase being requested, the triggers, and the recent phase history.

5.1.2.4 System software recovery. The system software recovery is done after the processor hardware has been configured, the permanent memory verified and initialized, and the peripheral hardware has been reconfigured. The SINT program performs this initialization of the transient memory which includes the saving of stable calls on phases 2 and 3. SINT is organized into a single control module and several specific initialization modules. The control module will select the appropriate initialization modules to be run depending upon the phase being requested and will then execute them in a predetermined order. Each initialization module performs a fairly self-contained initialization function such as zeroing all scratch areas or initializing the network maps in call store. When SINT completes the software initialization, it passes control back to SICO to prepare the system for restart.

5.2 Manual system recovery

Manual intervention may be necessary to regain system sanity or overcome system deficiencies. Before such action can be taken, one must be able to recognize the need for manual intervention. There are several types of indicators.

The master control console[6] has a number of visual displays that indicate system performance. If these alarm repeatedly, manual action should be taken. An excess of audits or interrupts can indicate system failure that requires manual intervention if the system does not initiate effective corrective action within a reasonable time (e.g., phase of initialization).

Once a need for manual action is ascertained, the correct manual action must be chosen from a number available. All combinations of processor configurations can be forced and tried if the processor is insane.

System initialization phases 1 through 4 can be activated manually with options that:

(*i*) Inhibit error indicators and force the system to ignore errors
(*ii*) Initialize short-term transient call store
(*iii*) Initialize long-term transient disk storage
(*iv*) Remove system data-base changes just activated

If the system is sane enough to perform input-output requests, a host of commands are available for manual intervention. All configurable hardware units in the system can be forced in and out of service, have their individual internal-error indicators inhibited and uninhibited, and have their internal memory and control points examined and modified via commands from a teletypewriter. Almost all actions that are performed automatically by the system can also be initiated by manual action such as demanding audits, requesting phases of initialization, or requesting diagnostics. Many additional actions that cannot be performed automatically can be executed by the craft people.

VI. SUMMARY

In order to meet the very stringent dependability and maintainability requirements, four very large software packages were developed in the area of maintenance software. The software-error recovery package detects and recovers from software malfunctions. The hardware-error recovery package recovers from hardware faults through fault detection and reconfiguration. Diagnostic programs are used to detect and locate hardware faults in a given faulty unit. The fourth package, system recovery, provides for overall coordinated system recovery from multiple or very severe hardware and software malfunctions. All four software areas were developed concurrently, each having to meet its own stringent area requirements as well as having to interface with the other maintenance software areas. The end result was a unified software package that covers the detection of and recovery from hardware, software, and system malfunctions both automatically and manually.

REFERENCES

1. R. W. Downing, J. S. Nowak, and L. S. Tuomenoksa, "No. 1 ESS Maintenance Plan," B.S.T.J., *43*, No. 5, Part 1 (September 1964), pp. 1988–2018.
2. E. J. Aitcheson and R. F. Cook, "No. 1 ESS ADF Maintenance Plan," B.S.T.J., *49*, No. 10 (December 1970), pp. 2831–2856.
3. P. W. Bowman, M. R. Dubman, F. M. Goetz, W. R. Hudgins, D. E. Lutz, and E. H. Stredde, "1A Processor: Maintenance Software." B.S.T.J., *56*, No. 2 (February 1977), pp. 255–288.
4. J. Janik, Jr., J. H. Huttenhoff, M. F. Slana, and F. H. Tendick, "Peripheral Hardware," B.S.T.J., this issue, pp. 1029–1065.
5. "LAMP: Logic Analyzer for Maintenance Planning", B.S.T.J., *53*, No. 8 (October 1974), pp. 1431–1555.
6. A. H. Budlong, B. G. DeLugish, S. M. Neville, J. L. Quinn, and F. W. Wendland, "1A Processor: Control System," B.S.T.J., *56*, No. 2 (February 1977), pp 135–180.